Melody: A Distributed Music-Sharing System

by

James Robertson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

[February 2003]
January 2003

Author .....
Department of Electrical Engineering and Computer Science
6 February, 2003

Certified by .....
M. Frans Kaashoek
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Melody: A Distributed Music-Sharing System

by

James Robertson

Submitted to the Department of Electrical Engineering and Computer Science
on 6 February, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Text search has been an important feature of file sharing applications since Napster. This thesis explores a directory hierarchy for categorizing and retrieving content in peer-to-peer systems as an alternative to keyword search. This thesis discusses Melody, a music sharing system that implements a directory hierarchy. Melody is built on Chord, a distributed lookup algorithm, and DHash, a distributed hash table. We evaluate the performance consequences and usability of Melody. In addition, this thesis presents two support applications: Autocat, to support automatic categorization, and Paynet, to encourage users to pay for the songs they listen to.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to thank Frans Kaashoek, my advisor, for his guidance and endless support. I also want to thank David Karger for his endless enthusiasm and ideas, such as the name "Melody." Russ Cox suggested the directory idea, which propelled this project in its fateful direction. Frank Dabek was essential in scrutinizing problems with Chord and DHash. Benjie Chen's insight into congestion control was also crucial.

Thank you to those who who believed in me, and anyone else I forgot.

Contents

Chapter 1

Introduction

Data retrieval in peer-to-peer systems is difficult because the amount of data is vast and ever-changing. Text search indexes have been used to catalogue and locate data in peer-to-peer file sharing systems. Such an index can be queried to discover the location of data, as well as discover the existence of data of interest.

An index that is distributed across many nodes in a peer-to-peer system is of special interest because of its potential to scale with the number of nodes in the system. Each node can now do part of the work necessary for answering search queries, which closely matches the philosophy of peer-to-peer systems providing services for their peers. However, a distributed index suffers from a variety of design problems such as how to split up the index, and how to query all the parts efficiently [11].

An alternative to peer-to-peer text search is a directory hierarchy. Directories can be used to categorize content. Each directory layer can separate content into finer and finer grained categories. This system has several advantages over text search, such as the ability to browse. It also has several disadvantages, such as the need to categorize content, which is difficult to do automatically.

These categories might be used as follows: the first level category might be genre, with each genre containing categories for artists in that genre, and each artist category can contain album categories, and so on. Users are able to browse through categories to find new music and can add new songs and directories through a web page interface. Figures 1-1 and 1-2 are screenshots of Melody's web interface.

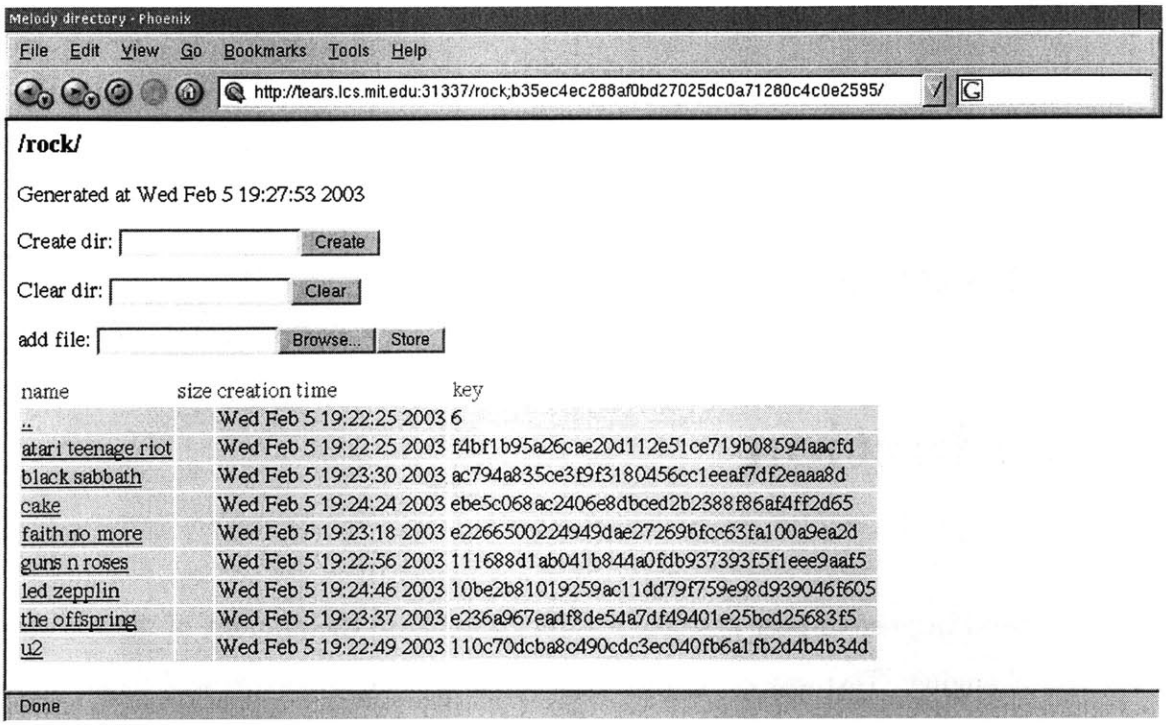

Figure 1-1: screenshot of web interface

Melody is an implementation of a directory-based music sharing application built on top of Chord [23] and DHash [5]. In Melody, the directory hierarchy is append-only to ease the implementation of the system as well as limit the damage that malicious users can do.

The following chapters cover all aspects of the Melody system. Chapter 2 covers the background of peer-to-peer systems and the Chord system that Melody is based on. Chapter 3 discusses the basic design and implementation of Melody's directory hierarchy. Chapter 4 evaluates the Melody system. Chapter 5 presents the automatic music categorization software used to support Melody. Chapter 6 discusses Paynet, a system for contributing money to artists. We present conclusions and future work in Chapter 7.

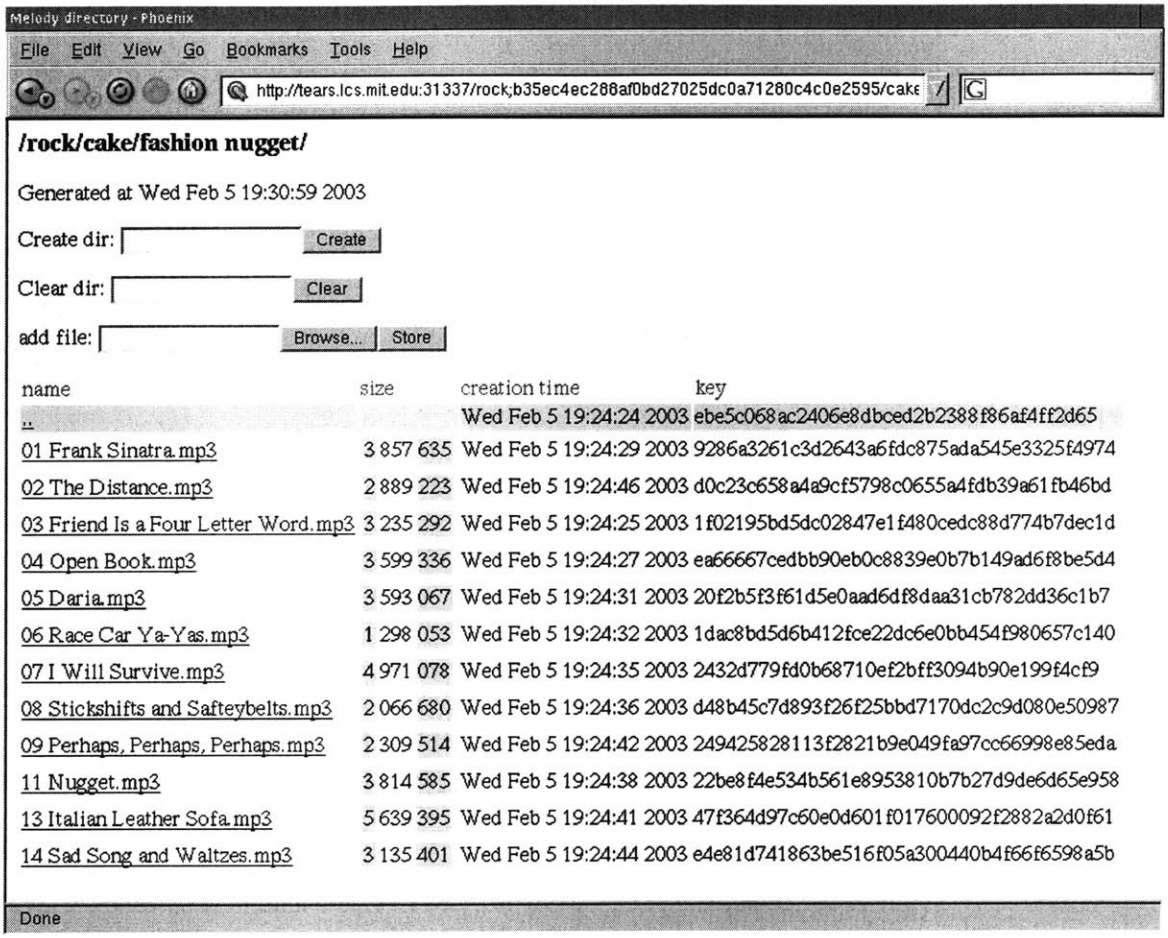

Figure 1-2: screenshot of web interface

Chapter 2

Background

2.1 Popular Peer-to-peer systems

Peer-to-peer systems were popularized by the success of music sharing systems like Napster [16], Gnutella [8] and Kazaa [10]. These music sharing systems share files, usually in MPEG Audio Layer-3 [15] (MP3) format, from the local user's personal collection. In these systems, files are automatically published when a node joins. In Kazaa, a node that has just joined must publish its index of files with its supernode.

Napster, Gnutella, Kazaa and other peer-to-peer music sharing service also provide a mechanism for finding and retrieving files. These systems use keyword searching rather than a directory hierarchy to locate files. Napster maintained a central index for searching for files. Kazaa and Gnutella have a distributed search mechanism. They flood queries to many nodes to try to find matches. Kazaa uses a kind of distributed indexing in the form of *supernodes*. These supernodes keep an index for some limited number of nearby normal nodes. Searches then only need to be flooded to the supernodes.

Searching on unstructured systems like Gnutella appears to be feasible only because most users are looking for "hay in a haystack" rather than a "needle in a haystack" [12]. The majority of users have the popular files, "hay," so a flooding query is likely to receive a quick response. Files that are temporarily unpopular may not be visible within parts of the network, or may disappear from the peer-to-peer

network if all the nodes that host these files leave. In contrast, a keyword search system based on a lookup algorithm such as Chord or Kademlia [13] supports looking for the "needle," a particular piece of data that can be found anywhere in the network. Unfortunately, most users do not know the exact name of what they want, so the "needle" model does not fit their needs well.

Overnet [18] is a peer-to-peer file sharing system built on Kademlia. Kademlia is an overlay routing network that permits the lookup of nodes through a unique ID. In Overnet, users publish an index of the files on their computer into a distributed index in Kademlia. Other users can search this index to discover content, and transfer it directly from the owner's computer. This is a mixture of the Kazaa and Melody models, because files are stored directly on the users' computers, but files are found through an overlay routing network. Public information on this system is limited, so further comparisons are difficult to make.

Freenet [4] is a peer-to-peer file storage system. It is similar to Melody in that content is published into the system as a whole, rather than just shared from a user's computer.

2.2 Chord and DHash

Melody uses DHash [5] to store and retrieve these files (see figure 2-1). Melody files are split into 8192-byte blocks. DHash is a distributed hash table which stores key-value pairs on nodes in a peer-to-peer system. In the case of Melody, the key is some identifier for the block, and the value is the block itself. DHash performs hash table lookups by using the Chord [23] lookup algorithm.

Chord is an overlay routing system that DHash uses to discover the location of keys. With Chord, each node is assigned a 160-bit address called a *Chord ID*, which is based on a SHA1 [6] hash of its IP address. This address determines where the node will be in the Chord ring, which is 160-bit circular address space. Each node knows the Chord ID and IP address of its predecessor and several of its successors in the ring. It also has a table of fingers, which is a list of a series of nodes that are

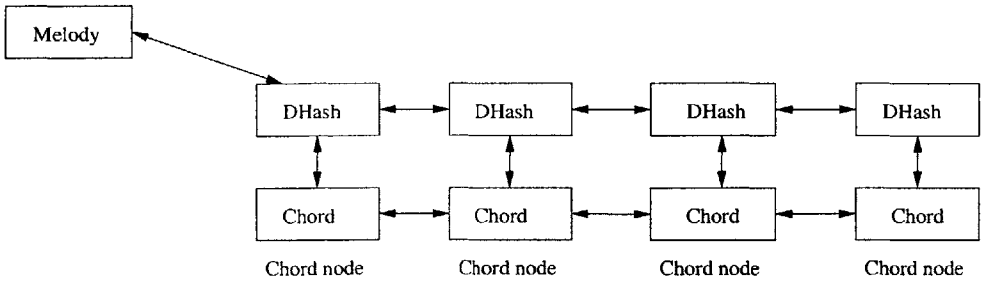

Figure 2-1: Melody and DHash/Chord Layers

at inverse powers of two around the ring. This list contains the node that is halfway around the ring from the current node, one quarter of the way around the ring, and so on. The finger table is used to quickly find nodes of interest by querying other nodes for closer entries in their finger tables.

Given a key, DHash uses Chord to find the node responsible for it in log(N) messages, where N is the number of nodes. This property permits DHash to scale to very large systems with little overhead. In DHash, keys also are 160-bit addresses, and their home node is the one that follows it in the Chord address space. The home node is responsible for replicating blocks on the nodes following the home node, so that in the event of a home node failure, a replica of the block already exists on the new home node. This is possible because in the event of nodes failing, Chord is robust enough to continue to locate the correct node.

The only ways for a block to leave DHash is for all the replicas to fail before any of them have a chance to create a new replica, or for the system to remove the block due to lack of space. With enough replicas and storage space, with high probability any particular block will be available from DHash.

A common type of block that is stored in DHash is called a *content-hash* block. This type of block stored under a key that is the SHA1 hash of its contents. This gives the data the property of being self-certifying. A user that retrieves a content-hash block from DHash can verify that it is the correct block by comparing the hash of its contents with the address used to retrieve it. This property makes it extremely difficult for malicious nodes to introduce incorrect data into the system. Content-hash blocks are immutable by their very nature, since changing them also changes

their address.

Another type of DHash block is the *public-key* block. It is a block that is signed using the private key of an asymmetric key pair. This signature is stored in the block, along with the public key. The block is stored in DHash under a key that is a hash of the public key. The user can use the public key to verify the signature, as well as the address that the block is stored at. The owner of the private key can store an updated version of the block, along with an updated signature. This makes the block mutable, as well as self-certifying.

Melody uses another type of block which is not self-certifying. This type is called an *append* block. An append block exists at a fixed address, and is mutable, but it only supports an append operation for changing its contents. The uses and security implications of an append block are discussed in Chapter 3.

Chapter 3

Directory Hierarchy

In Melody, users find content by interacting with a directory hierarchy. They are presented with a list of genre categories. Choosing one of the genres leads to a list of artist categories. Here, the user can create new artist directories. Choosing one of the artists leads to a list of albums. Here, the user can add new albums. Choosing one of the albums presents a list of music files to the user. Here the user can choose an individual song to listen to, or add new songs.

The content being added needs to be added to the appropriate category if it is to be easily found. The directory hierarchy system is only as useful as its categories are descriptive, which depends on the content being categorized properly. Since categorization is a tedious task for users to do manually, we investigated the feasibility of automatically categorizing MP3 files. We constructed an automatic categorizer called Autocat which is discussed further in Chapter 5.

Properly categorized songs provides an accurate source of song names that can be used for other purposes, such as paying artists. We propose Paynet, which is a system to voluntarily contribute money to artists on a usage basis. We conjecture that users are willing to pay artists for content as long as it is convenient to do so. Paynet takes advantage of the readily available and consistent names in Melody's directory hierarchy. Paynet's operation is discussed in Chapter 6.

The rest of this chapter will discuss the design and implementation of Melody's directory hierarchy, as well as discuss the user interaction and security implications

of its use.

3.1 Design

A directory hierarchy is an alternative to distributed indexing. A directory hierarchy is composed of directory blocks, which consist of directory entries. Each directory entry associates a name with an identifier for another directory or piece of content in the system. A hierarchy can then be built, with directories referring to other directories. Users can start a query at a root directory, which has a well-known identifier.

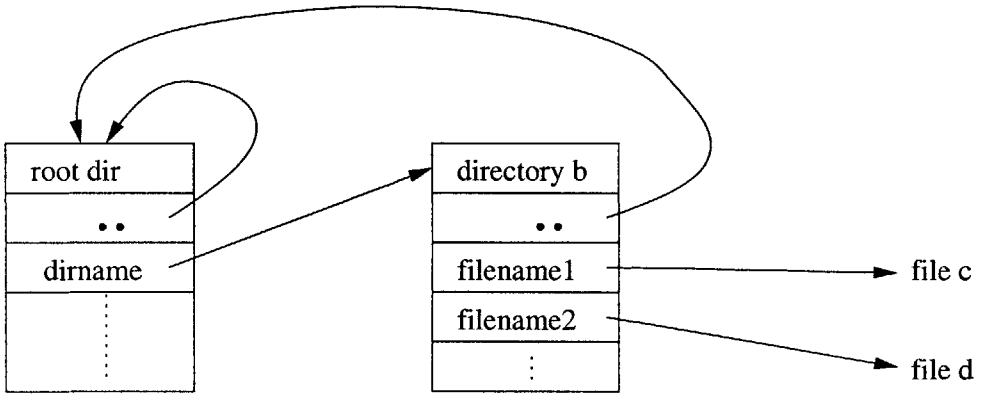

Figure 3-1: Simple directory hierarchy

This directory hierarchy is created by users adding entries and new directories to existing directory blocks. This depends on directory blocks being mutable.

3.1.1 Directory Blocks

Directory blocks are single blocks which are lists of directory entries. Each directory block represents a directory, and each entry represents a subdirectory or file in that directory. Each directory entry has a name, some metadata, and a content hash or key. This content hash or key is the reference to the music content or directory block associated with this name. This key is also appended to the name, to keep each name unique and identifiable even if two different files with the same name are added to the same directory.

Directory blocks are implemented as a new type of DHash block called an *append* block. There are stored in DHash so that they can be distributed and replicated like the other data blocks in the system. This additional type of block is needed because other DHash blocks are either immutable, such as content-hash blocks, or cannot be changed without a read-modify-write cycle, such as public-key blocks. Having to read the block before modifying it would require some kind of locking to avoid the race condition that could occur with multiple writers. Instead, the node responsible for an append block can accept RPCs that contain data to be added to the end of the block. Append blocks are easier to implement than lockable mutable blocks because it avoids the whole issue of acquiring, releasing, and possibly revoking locks.

A system implemented with append blocks is not as flexible as one implemented with some kind of lockable mutable block or public-key blocks. However, using append blocks reduces the complexity of the system significantly. Furthermore, append is sufficient to make interesting applications, such as Melody.

Directory blocks need to be stored at fixed IDs in the DHT so that other directory blocks can refer to them. Using a content-hash as an ID is precluded because the directories need to be mutable. Instead, Melody uses the hash of a public-key generated by the user that created that directory. The user also stores his public key in the append block when he creates it. That user then owns the directory block, and has the unique ability to clear or replace its contents, although anyone can append data to it.

This clearing request is authorized by an accompanying signature. This signature is generated using the user's private key, and a random number that the home node adds to the append block when it is created. The home node can then verify that this signature by using the public key in the append block. This confirms that the clearing request came from the user that owns the corresponding private key.

Directory entries are the chunks of data that are appended to directory blocks. When users of the system are creating new files and subdirectories, Melody appends new directory entries to a directory block. This append-only property means that files cannot be modified after being created, and directories cannot have entries removed,

except by the owner of the block.

3.1.2 Content Blocks

Some directory entries refer to content. Content can be music files, such as MP3 files, or other media that can be categorized. Melody stores all files using content-hash blocks, and uses a scheme to keep track of the blocks that is similar to the block storage scheme used by the Venti [21] storage system at Bell Labs.

The files are referred to using a 160-bit Chord ID. This ID is the content-hash of a content-hash block that is the root of a tree of content-hash blocks. It contains the IDs of other content-hash blocks. These other blocks can either be data blocks, or can contain more IDs. There can be several layers of these ID-containing blocks, which we refer to as venti blocks. This enables the system to support files of unlimited size by increasing the depth of the tree of venti-blocks.

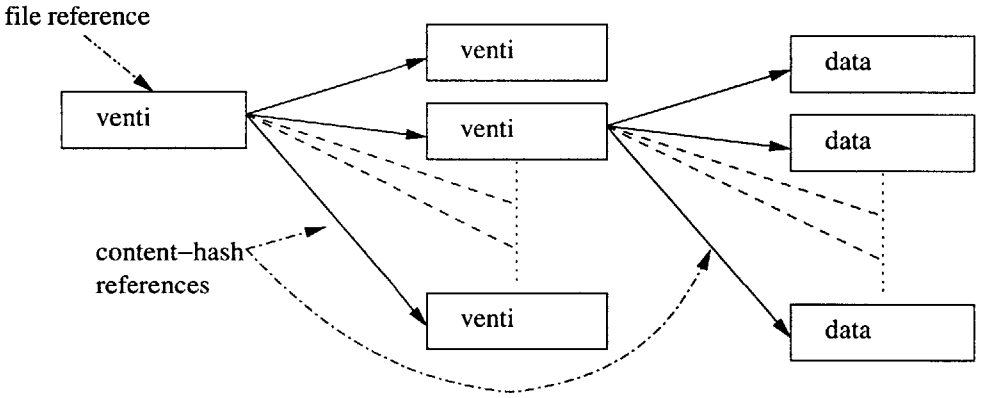

Figure 3-2: Tree of venti blocks used to keep track of files data blocks

The use of content-hash blocks means that content can be added to the system, but not changed. Also, when files are added that are identical to existing files in the system, no additional storage is needed because the addresses of the new blocks will refer to existing content-hash blocks. This choice has the benefit of not penalizing users for inserting content multiple times. Finally, since the blocks are use a content-hash for their key with an essentially random distribution, the storing a file causes its blocks to be stored at a many different nodes around the Chord ring.

3.1.3 User Interaction

In terms of user interaction, the advantage of a Yahoo-like category model over a Google-like searching model is that the system presents to the user precisely what artists and songs are in the system, rather than having the user try to guess keywords that are in the name of the music. Users can browse through artist or genre categories of music that is similar to music that they know they like. Also, a user does not have to have an exact query in mind to make progress when looking for content. Instead, they can browse through categories and make local decisions to try to bring them closer to their goal. For example, they might be looking for a song from an 80's European pop band whose name begins with 'c', and with the proper categories they should be able to browse and find that band.

A Google-like searching model is prone to returning false positives for files that match the keyword in properties that the user did not intend to search. However, a keyword search has the advantage that it can find disparate content that would be in vastly different categories in directory hierarchy.

3.1.4 Security

The directory hierarchy scheme has several security problems, mainly to do with trusting the directory hierarchy.

Trusting Nodes

Nodes cannot be trusted to return correct directory blocks. In a simple scheme, each directory block is stored on a single node, with some replicas for reliability. A malicious node could serve incorrect or subtly changed directory blocks. Unlike other data blocks, these directory blocks cannot be self-certified, because they depend on being found at a fixed identifier even when their content changes. Clients can only check to see if the format of the directory block makes sense. However, potentially a quorum scheme [3] could be used to allow clients to compare and verify directory blocks. This would provide some measure of verifiability.

Another form of verifiability is implementing some of the directory blocks using public-key blocks. This would allow clients to certify that these blocks have not been changed by a malicious node. Since these blocks would only be mutable by a trusted few, this scheme would be practical only for blocks that change infrequently, such as blocks near the root of the directory tree. These also happen to be the most important blocks in the directory tree, since a bogus version of it has the greatest effect on a client's view of the directory hierarchy.

Trusting Users

Malicious users are another threat to directory blocks. The directories in the hierarchy must be mutable by anyone to permit the addition of new data to the system. This is much more dangerous than the previous attack because anyone can modify directory blocks, not just the node responsible for the block. So a malicious user has a much larger theater of action. The damage is limited by the append-only nature of the directory blocks. A malicious user cannot remove correct directory entries, only add incorrect entries or garbage. This attack will not prevent a user from retrieving valid entries, but only slow them down. The data in the system previously will still be available. The owner of a directory block has the unique ability to clear it, which can be used as last-ditch measure to save a directory after garbage has been added to it. Ideally, after clearing it, the owner of that directory block would immediately add the "good" data back into that directory.

Security is a problem whenever you have data that is modifiable by anyone. Although this directory hierarchy does not offer many guarantees about the robustness of the directory blocks, smart clients may be able to skip over or ignore incorrect or invalid entries. The append-only blocks help to maintain access to data in the system, but this comes at the cost of losing the ability to delete or modify records.

Furthermore, users are used to dealing with security problems, given the experience of existing peer-to-peer systems. Systems like Gnutella contain many files that are mislabeled by users, either accidentally or maliciously. Since these titles do not reflect their content, the benefit of searching is reduced. In response, some systems,

such as Kazaa, have implemented moderation to their file listings. A search result can return a ranking associated with each file found. These rankings are set by users in accordance with their happiness with the file's quality and accuracy of its title. However, not all files are ranked.

3.2 Implementation

`Melody` is implemented as a gateway that gives the user access to data stored in a Chord ring. To communicate with the user, `melody` is implemented as a web-proxy in C++ using the `async` library from SFS [14]. It provides the user with a web interface where each web page represents a directory in Melody's directory hierarchy. Links on the web page refer to other directories or files stored in Melody. When a user clicks on a link, the appropriate directory or file is fetched from DHash on the Chord ring, and presented to the user. Melody communicates to the Chord ring over a socket file to an `lsd` process.

`Lsd` is an implementation of Chord and DHash in C++ also using the `async` library. Each physical node in the network runs an `lsd` process. Each `lsd` process is one or more virtual Chord nodes in a Chord ring. The `lsd` programs only need to be told of one other existing node to join a Chord ring. They communicate with each other over the network using Sun RPCs [22].

Melody uses only a few RPCs to communicate with `lsd`:

RPC name	description
`insert`	Inserts a block into Chord using its content-hash as its key.
`retrieve`	Given a key, retrieve either an append or content-hash block.
`append`	Creates or appends data to an append block in Chord.

These RPCs are sufficient for Melody to store and retrieve more complex data structures, such as files and directories, in DHash. Blocks are only retrieved as complete entities. Partial retrieves are not supported. In order to mask latency from the user when retrieving files, Melody reads ahead and tries to maintain 50 outstanding

block retrieve requests.

Append is used to add keys to venti blocks, and to add directory entries to directory blocks. Both are appended one at a time. The directory entry format is listed in figure 3-3. Directories are implemented as a single layer venti structure so that a directory's directory blocks remain smaller than 8192-bytes. The ability to clear append blocks is implemented as a modification to the append RPC, because Melody always wants a directory block to have some initial content, such as a directory entry for the parent directory.

```
typedef struct {
  int type;
  char key[20];
  int size;
  struct timeval ctime;
  char entry[256];
} dir_record;
```

Figure 3-3: Directory entry format

Chapter 4

Evaluation

We evaluated Melody in several different dimensions. We evaluated Melody to see if its transfer speed can support the realtime streaming of MP3 files to software player. Unlike traditional peer-to-peer models, nodes in Melody do not store complete files. This forces players to retrieve blocks from a variety of different nodes. Since this involves contacting several different computers, it is important to evaluate whether this harms streaming playback performance. We also tested Melody to see if the content was still available after node failures. Finally, the storage requirements are analyzed to examine their overhead.

4.1 Methodology

We tested Melody on on PlanetLab [20], which is a test-bed for experimenting on peer-to-peer and other distributed networking technologies. It is composed of similar computers at various research centers around the world. We used a subset of the approximately 100 production nodes in PlanetLab for testing Melody. See table 4.2 for the list of nodes used. On each node, we ran an `lsd` process (see Chapter 3). A single instance of `melody` runs on the limited-enthusiasm.lcs.mit.edu node, and communicates with `lsd` to insert and retrieve data from DHash. A tool for inserting complete directory hierarchies into Melody, called `insertdir.pl`, is used to test the transfer rate of data onto Chord. The `wget` program is used to retrieve the directory

hierarchy from Melody's web interface.

A simplified MP3 file model is used to evaluate the adequacy of file transfer rate. In this model, the MP3 files are of a constant bit-rate of 128 kilobits per second. This corresponds to a required transfer rate of 16 kilobytes per second. Although some MP3 files are recorded at high bit-rates and also variable bit-rates, the most common average bit-rate is around 128 kilobits per second.

We evaluated the speed and robustness using a pre-created directory hierarchy that consisted of 152 megabytes of MP3 files taken from CDs by "The Beatles." The experiment consisted of several timed retrieve and insertion tests using Melody on a Chord ring with blocks replicated across 3 nodes and later replicated across 5 nodes. The Chord rings were composed of PlanetLab nodes listed in table 4.2, and were restarted with clean storage before every insert run.

4.2 Speed

The retrievals took vastly varying amounts of time. The first retrieval occurred with all the nodes listed in table 4.2 running. The second occurred with the nodes listed in table 4.3, which is approximately one-third of the ring. The insertion test is labeled "insert," and the two retrieval runs are labeled "retrieve 1" and "retrieve 2" in table 4.1. Each row consists of a insert and retrieve test run on a single Chord ring. Early testing was performed with `lsd` replicating blocks across three nodes. These test runs correspond to the first row in the table. Two partial tests were run with blocks being replicated across five nodes for improved reliability. These test runs correspond to the second and third rows in the table. The data for these last 2 runs is incomplete due to testing accidents.

In the first test run, retrieving 152 megabytes of data took 63 minutes. This equates to a transfer rate of 39.9 kilobytes per second. In addition, closer examination of the transfer logs shows that none of the transfers was slower than 28 kilobytes per second. These rates are adequate to support the streaming MP3 files in real-time.

The transfer speed with three replicas is also entirely adequate to support the

replicas	operation		
	insert	retrieve 1	retrieve 2
3	28:33	63:26	244:34
5	20:23	-	-
5	136:46	236:01	-

Table 4.1: Insert and retrieval times in minutes:seconds

insertion of large directory hierarchies. At this rate, a user can insert average bit-rate MP3 files roughly five times faster than he could listen to them. Although this is not lightning fast, it is not a huge detriment since it is a one-time cost. The insertion speed for the second test run is even faster than the first.

However, some tests took much longer. The second retrieval of the first test run finished in 244 minutes, which is a transfer rate of 10.4 kilobytes per second. The third test run was also slow. The transfer speeds of these test runs are not adequate.

This vast difference in transfer speeds is surprising, since little is changing in the Chord ring between runs.. Both the addresses of the Chord nodes and the addresses of the file data blocks are the same between test runs, so the variance cannot be blamed on radically different Chord network configurations. The addresses of the directory blocks are different between each run, because their address is based on the hash of a randomly generated public key. However, the latency for accessing directory blocks accounts for only a small part of the total transfer time. This speed difference also cannot be solely blamed on the extra network traffic required to create two more replicas, because the second test run was faster than any of the others. There is some other factor causing some nodes to have much higher latencies when responding to RPCs, which we explore next.

4.2.1 Chord node latency

To examine these large variances further, additional tests were constructed to examine differences in the PlanetLab network. There are two major differences between running a Chord ring on a local testbed and running it on PlanetLab. One is the

planetlab2.cs.arizona.edu	93.651 ms
planlab2.cs.caltech.edu	92.867 ms
planetlab3.xeno.cl.cam.ac.uk	80.885 ms
planetlab-3.cmcl.cs.cmu.edu	27.018 ms
planetlab3.comet.columbia.edu	17.046 ms
planetlab2.cs.duke.edu	25.766 ms
planet1.cc.gt.atl.ga.us	32.329 ms
righthand.eecs.harvard.edu	4.338 ms
planet2.berkeley.intel-research.net	88.474 ms
planet3.seattle.intel-research.net	78.296 ms
planetlab2.ext.postel.org	96.957 ms
planetlab2.netlab.uky.edu	46.769 ms
planetlab2.cs-ipv6.lancs.ac.uk	129.503 ms
planetlab2.eecs.umich.edu	43.942 ms
planetlab1.lcs.mit.edu	0.597 ms
planetlab2.lcs.mit.edu	0.689 ms
planet2.scs.cs.nyu.edu	10.276 ms
planetlab-2.cs.princeton.edu	19.222 ms
ricepl-3.cs.rice.edu	48.719 ms
planetlab-2.stanford.edu	79.732 ms
pl1.cs.utk.edu	32.111 ms
planetlab2.cs.ubc.ca	86.550 ms
planetlab1.cs.ucla.edu	122.747 ms
planetlab3.ucsd.edu	90.517 ms
planet2.cs.ucsb.edu	89.578 ms
planetlab2.cs.unibo.it	113.772 ms
planetlab2.it.uts.edu.au	240.399 ms
planetlab3.csres.utexas.edu	49.848 ms
planetlab2.cis.upenn.edu	20.723 ms
planetlab3.flux.utah.edu	62.689 ms
vn2.cs.wustl.edu	42.425 ms
planetlab2.cs.wisc.edu	43.360 ms
limited-enthusiasm.lcs.mit.edu	0.026 ms

Table 4.2: List of PlanetLab nodes used for Insertion and Retrieval 1 test, along with average of 4 pings

planetlab2.cs.arizona.edu
planlab2.cs.caltech.edu
planetlab3.xeno.cl.cam.ac.uk
planetlab-3.cmcl.cs.cmu.edu
planetlab2.cs.duke.edu
planet1.cc.gt.atl.ga.us
planetlab2.netlab.uky.edu
planetlab2.cs-ipv6.lancs.ac.uk
planetlab2.eecs.umich.edu
planetlab1.lcs.mit.edu
planetlab2.lcs.mit.edu
planetlab-2.cs.princeton.edu
ricepl-3.cs.rice.edu
planetlab-2.stanford.edu
pl1.cs.utk.edu
planetlab2.cs.ubc.ca
planetlab3.ucsd.edu
planet2.cs.ucsb.edu
planetlab2.cs.unibo.it
planetlab2.it.uts.edu.au
vn2.cs.wustl.edu
planetlab2.cs.wisc.edu
limited-enthusiasm.lcs.mit.edu

Table 4.3: Nodes used for Retrieval 2

limited-enthusiasm.lcs.mit.edu
planetlab2.comet.columbia.edu
planetlab2.cs.duke.edu
righthand.eecs.harvard.edu
planet2.scs.cs.nyu.edu
planetlab-2.cs.princeton.edu
planetlab2.cs.umass.edu
planetlab2.cis.upenn.edu
planetlab-2.cmcl.cs.cmu.edu
planetlab3.xeno.cl.cam.ac.uk

Table 4.4: Nodes used for network latency test

difference in network latency between the various nodes. See table 4.2 for a listing of ping times from MIT. The other is the difference in bandwidth between the sites. Some sites, such as CMU, have a 10 megabit link rather than a 100 megabit link. The computers at the sites are all similar, so the network differences are the key factors.

These tests used a small Chord ring composed of the nodes listed in table 4.4. This ring focused on highly connected nodes on the east coast of the United States, along with a node with much larger ping time in the United Kingdom, and a node with much less bandwidth located at CMU. By leaving out the CMU node, the effect of well connected nodes communicating with a high network latency node could be examined. By leaving out the UK node, the effect of well connected nodes communicating with a low bandwidth node could be examined. Leaving out both oddball nodes is useful for a baseline. Chord IDs were chosen that evened out the spacing between nodes, such that the UK and CMU nodes got an even amount of storage and replica storage RPC traffic.

A Chord benchmarking program called `dbm` was used to store and fetch 8 megabytes of data in the form of content-hash blocks. The number of fetch operations in flight was varied to examine how much pipelining could hide network latencies. Tests were run 3 times for consistency (see table 4.5).

operation	baseline	with CMU	with UK
store, 10 operations in flight	19.85	20.01	24.33
fetch, 10 operations in flight	5.69	5.68	19.93
fetch, 10 operations in flight	5.30	5.21	18.33
fetch, 10 operations in flight	4.95	6.07	18.71
fetch, 50 operations in flight	4.89	4.14	10.16
fetch, 50 operations in flight	4.11	4.77	9.53
fetch, 50 operations in flight	4.25	4.49	10.13

Table 4.5: `dbm` store and fetch time in seconds

The results demonstrate that network latency is a much more important consideration than bandwidth for Chord rings. Even though the RPCs are asynchronous, an application may be difficult to have enough operations in flight to completely mask

the latency.

To examine the effect of which transport the RPC layer used, a second test run was performed using TCP instead of Chord's UDP with congestion control. The times for this test run are listed in table 4.6.

operation	baseline	with CMU	with UK
store, 10 operations in flight	16.12	16.111	16.30
fetch, 10 operations in flight	9.69	10.01	11.95
fetch, 10 operations in flight	10.43	9.75	12.41
fetch, 10 operations in flight	8.21	10.43	11.48
fetch, 50 operations in flight	3.21	3.69	4.14
fetch, 50 operations in flight	2.85	3.76	3.91
fetch, 50 operations in flight	3.34	3.27	3.95

Table 4.6: `dbm` store and fetch time in seconds, TCP transport

These figures exhibit much less variance than the first latency test run, and demonstrate a problem with Chord's default congestion control. Examination of the timings between packets when running with Chord's default RPC transport shows that the congestion control is inadequate when several well connected nodes are interacting with a high latency node. If an RPC packet is dropped by the network, then other packets waiting in the sender's window are not sent until the dropped RPC is resolved. This stalls all RPCs waiting to be sent. This effect is exacerbated for nodes with higher network latency. This is the ultimate root cause of the vast variance in latencies, because the effect is intermittent and depends on packets being dropped.

The TCP transport is impractical for large systems because it maintains an open connection between every node. However, it is useful as a demonstration of how congestion control could work if it used more information. Chord's congestion control needs to be improved and separated from the RPC retransmit mechanism.

The system does have the potential to perform well, even when blocks are stored on three different continents. Perhaps additional replicas would make it possible for Melody to avoid retrieving from slow nodes, by selecting the fast replicas, thereby improving transfer performance.

4.3 Robustness

This test involved stopping some nodes and retrieving data from the system. Robustness is demonstrated through the continued availability of all the content in the system. In the second retrieval tests, all the MP3 files remained available even after one-third the nodes had been stopped. Examining the content retrieved showed that it was retrieved without corruption or missing data. In other robustness tests with smaller rings, all the content was still available after two-thirds of the nodes were be stopped.

Nodes were stopped sequentially in order to give the remaining nodes time to maintain replicas of the blocks. Stopping too many nodes at once has the potential to remove all the replicas of a particular block. The chance of all the replicas being removed goes down as the number of replicas increases, so for large systems with high turnover rate more replicas will be needed.

4.4 Storage

Melody's directory hierarchy has low storage requirements. Directories are formed of one Venti-style block, which refers to many directory-entry blocks. Each directory entry contains a path name component that is 256-bytes long, a 160-bit ID, and some other metadata. Overall, each directory entry is 292-bytes long. If each file is at an average depth of 4, and each level has 50, then the overhead per file is about 1.02 directory entries, which corresponds to about 298-bytes per file.

Melody stores files with even less overhead. In general, the overhead is only that of a 160-bit ID per block added to a venti block. With a block size of 8192-bytes, this overhead is less than 0.24%. For very large files, this increases slightly because there are multiple layers of venti blocks, but the increase is too small to bother calculating.

The overall overhead is then about .0024 times the file size plus 300-bytes, which is miniscule. The actual overhead is much greater, however. The current implementation of DHash in the `lsd` program stores all blocks in a Berkeley database. In actual

testing, storing 3 gigabytes of data resulting in 5 gigabytes of space taken up by the database files. This is most likely due to Melody's blocks being slightly larger than Berkeley DB's fixed internal blocksize. This would cause two internal Berkeley DB blocks, one of which is mostly wasted, to be used to store one Melody block.

Chapter 5

Automatic categorization

Melody must categorize songs properly for the system to be useful. Manual categorization of songs is a tedious task for users, since it involves making repetitive decisions. Computers are good at repetitive work, but poor at understanding the inconsistencies of human language. Unfortunately, humans have named most MP3 files. These names are usually identifiable by people, but sometimes inconsistent enough to foil simple computer pattern matching. Autocat attempts to automatically categorize songs by genre, artist and album.

It tries to determine the artist, genre and album of arbitrary MP3 files using the ID3 tag or filename. Autocat uses this information to insert songs into Melody's directory hierarchy in the form `/genre/artist/album/`. When album, artist and genre information is not available, Autocat uses "unknown" for any missing artist and album information, and "misc" for any missing genre information.

Although the task of automatically categorizing MP3 files is non-trivial, it does not have to be completely accurate either. Users of Melody have the opportunity to re-categorize files that appear to be in the incorrect place. Songs that are in the wrong genre are harder to find than songs with a misspelled name. This observation implies it is most important to have accurate genre categorization.

5.1 ID3

MP3 files often include metadata called ID3 tags. These tags are often created using information that people type in while listening to the song with their MP3 playing software. This can result in MP3 files with inaccurate or inconsistent naming schemes. These tags are also created automatically by software that queries CD databases during the MP3 encoding process. Although automatically tagged MP3 files often have much less variation than MP3 files that are tagged individually by people, the CD databases are also created by people, and so they can also have inaccuracies.

The ID3 tag specification has two major versions, ID3v1 and ID3v2. Version one has four fixed length text fields for artist, title, album and comment information, as well as a one-byte field for genre and a four-byte field for year. The text fields are limited to strings no longer than 30 characters. The one-byte genre field chooses a genre from a predefined table. ID3v1 tags are found at the end of MP3 files. Version two is much more flexible and allows an arbitrary number and order and length of text fields. ID3v2 tags are found at the beginning of MP3 files. Some MP3 files have both types of tags.

Autocat tries to use the ID3 tag information to help categorize MP3 files. The ID3 tag's artist, title, album and genre fields, are ideal for Autocat's purposes because they can be read without using heuristics to determine the fields, unlike filenames. However, their existence and ease of acquisition does not guarantee their accuracy. In addition, MP3 files with both ID3v1 and ID3v2 tags may have conflicting information contained in those tags.

The limitations of ID3v1 cause some inaccuracy in determining the artist, title and album. These inaccuracies due to the text fields being limited to strings no longer than 30 characters. If a string is exactly 30 characters long, Autocat has to assume it is truncated. A truncated name is not preferred, so in this case Autocat uses the corresponding name acquired from the MP3's filename instead.

Another limitation of ID3v1 tag format is the genre byte. The table that this byte indexes into has a limited number of genres, which do not fit the needs of everyone.

For example, it does not contain a "none" or "undecided" entry. When people fill in the artist and title fields, but do not choose a genre for an ID3v1 tag, the software that is creating that tag usually defaults to the first entry in the table, which is "Blues." It is more likely that a song's genre has been chosen incorrectly than for it to actually be a blues song. When testing Autocat, less than 11% of the songs tagged as "Blues" were actually blues songs, and so Autocat ignores the genre from the ID3v1 tag if it is "Blues."

ID3v2 does not have an problem with a limited choice of genres because it includes an optional text field for genre. This field permits much more flexible and complete naming, and it also allows ID3 tag creation software to not create a genre field if the user does not fill in any information. MP3 files can then be tagged without a genre field rather than an inaccurate one. However, this flexibility also means that misspelled and arbitrary genres, such as "genre," are sometimes created by ignorant users or software. Autocat ignores genres of the type "genre," "other," and others.

5.2 Filename parsing

Autocat also tries to interpret the filename to discover the artist and title. Autocat prefers to use the ID3 tag information if it is available, on the basis that if someone went to the trouble to add it, than it is more likely to be correct. Filename parsing is useful if the ID3 tag cannot be used.

Autocat uses Perl regular expressions to interpret the filename. It assumes the filename is of one of the following forms, in order of decreasing priority:

```
artist feat ignored - title.mp3
01 - artist - title.mp3
(artist) title.mp3
artist - title.mp3
title.mp3
```

Once a match is found, the rest of the filename forms are not tried. The last form ensures that at least some kind of title is found.

5.3 Freedb and Google

If Autocat has not been able to determine the genre either due to inaccurate or missing ID3 information, then it consults two other sources over the Internet. The first is a pair of databases constructed from Freedb [7]. The second is Google Directory [9].

5.3.1 Freedb

Freedb is a text database available at http://www.freedb.org/ of CD track names and disc info. It consists of separate text files for each CD which contain disc title, song title, artist and sometimes genre information. The disc's unique id number is used to name each file, resulting in a database indexed by disc id.

Autocat needs to discover genre by using either the artist, if available, or the title. Since Freedb is not indexed by artist or title, a perl script creates two new databases from the Freedb information. One of these new databases is indexed by artist, and the other is indexed by title. Both are in Berkeley DB [2] format.

For the song title database, the records contain the disc title, genre and artist. Because many songs have the same name, Autocat scans through all the matches for a given song title, and filters the records to remove other artists. The artist database is used in a similar manner, but with records that contain song title instead of artist.

5.3.2 Google Directory

Google Directory is a Yahoo-categories-like web site which contains categories of links to other web pages. Some of these categories are music related, and contain categories that include bands and artists organized by genre. Autocat uses another program called "`googleit`" to fetch and interpret Google Directory web pages.

For songs that do not match any entries in the database, `googleit` queries Google Directory using the artist's name. It then looks to see if any Google Directory categories match the artist's name. If they do, `googleit` retrieves the category's web page from Google, and searches it for any links to genre categories. This produces a list of potential genres. In the case of multiple reported genres, `googleit` generally

uses the last reported genre as the genre of the song. However, some of these genres are of the form "1990s" or "2000s," which are less useful in categorizing songs, so `googleit` prioritizes the genres and prefers to return non-year genres.

5.4 Complete method

The order of actions in Autocat is as follows: parse the filename into title and artist, if available. Look for an ID3 tag, and if available override the title and artist chosen from the first step. If no genre has been found, attempt to lookup the title in Freedb, then the artist in Freedb. The resulting Freedb record will contain a genre. Otherwise, lookup the artist in Google Directory and look for a genre in the category listings.

Several alternate courses of action are tried if neither the Freedb and Google lookups are successful in producing a genre. If only the ID3 tag was used to determine the artist and title, then Autocat retries the Freedb and Google lookups using the filename as the definitive source of artist and title. If both of these have been tried, and if the filename is of the form "`ignored - artist - title.mp3`", then this alternative filename parsing is tried.

Once the genre, artist, and any album name are determined by Autocat, then a C++ program called "`it`" is used to connect to Melody's web interface, lookup the appropriate directories and create them if needed, and upload the MP3 file.

5.5 Experimental Results

Autocat's most important metric is how well it categories songs. This criteria is hard to quantify, because a song's genre is sometimes a matter of taste. However, we can measure the number of songs that Autocat either categorized into "misc" or grossly miscategorized. We performed two test runs, one with a directory of 776 MP3 files, and another with a directory of 730 MP3 files.

From table 5.1, it is apparent that Autocat is able to categorize the vast majority of MP3 files. It has a success rate of greater than 80%. Examining the badly categorized

directory	A	B
number of files	776	730
time to categorize (average of 2 runs)	3m21s	4m36s
filenames successfully parsed	736	580
genre from ID3	414	316
genre from Freedb	139	144
genre from Google	159	184
uncategorized files	64	86
badly categorized	91	55

Table 5.1: Autocat test numbers

data more closely reveals the biggest offender to be ID3 tags with bad genres. This is not surprising because Autocat trusts the ID3 tag above all other sources. Although Autocat does not perform perfectly, it is suitable for initially sorting music that can later be better categorized by people.

Chapter 6

Paynet

Paynet is a system for donating money to artists. It consists of a central clearinghouse which users can pay money to, for example, $10 per month through Paypal [19]. As the user listens to music over the course of that month, their music playing software gathers statistics on the relative length of time that the user listened to each artist. At the end of the month, the music playing software sends this report to the clearinghouse. The clearinghouse can then allocate their money to the artists, again through Paypal, in proportion to how long the user listened to them.

This system has the benefit of being mostly transparent to the user. The user does not have to do any action per song to contribute money to the artist. Yet, the payments from the user to a particular artist can be of very small granularity, for example, a few cents, because the clearinghouse can coalesce many other payments together before giving the money to the artist.

6.1 Implementation

Paynet is implemented as a pair of programs. `Donation_plugin` runs on the user's computer, collects the statistics on their music listening habits, and sends them to the clearinghouse along with their payment information. Clearinghouse processes the statistics and does the accounting to pay the appropriate artist.

6.1.1 Music player monitoring

`Donation_plugin` is implemented as a plugin for XMMS [24]. XMMS is an music playing program for Unix that plays a variety of music file formats, including MP3 files. `Donation_plugin` queries XMMS for the currently playing song every second. This polling method is used because there is no API for informing the plugin when the music stops or changes. `Donation_plugin` then stores this information in a hash table of Berkeley database format so that the statistics are persistent in the event of reboots or crashes.

6.1.2 Messages

At a regular interval, for example every month, `donation_plugin` contacts the clearinghouse. It sends a donation message to the clearinghouse that includes the time period that the statistics were gathered over, payment amount, payment information, the source of each song, and the number of seconds that song was played for. The payment information is currently a Paypal id, but this field could easily be extended to handle credit card numbers or other forms of payment. The source of each song is either a pathname of the file, or a URL.

Currently, the donation message is sent out in the clear, but this is not suitable for a production system because it contains the user's payment information. To protect it, `donation_plugin` could use Secure Socket Layer (SSL) [17] to communicate with the clearinghouse. SSL encrypts communications as well as provides a certificate mechanism for authenticating servers. It is commonly used with web servers to protect customers that are making purchases. This is a similar scenario, so SSL is a good fit.

The user that sends this message does not need to be authenticated to the server, because the message already contains all the information needed to perform a donation. The payment information and the artists to donate to are both included in the message. This precludes the need to link this message with an existing identity stored on the server.

The clearinghouse responds with a unique transaction id that the user can use to

confirm that proper payment took place.

6.1.3 Clearinghouse

Clearinghouse is the program that coalesces donations into payments to artists. It decrypts each donation message, and looks at the list of songs for artists that it can identify in the URLs. It then computes the proportion of the payment amount that should be sent to each artist, and bills the user and credits the artists accordingly. This financial information is recorded in a Berkeley DB. The clearinghouse program currently does not do any actual billing with real money. However, it would be easy to query the database every month and generate the necessary transactions.

6.2 Trust

Trust is an important issue whenever money is involved. In this case, in order for the payment to go to the correct music artist, both the user and the clearinghouse have to operate correctly. It is difficult for the user to verify that the clearinghouse operating correctly without extensive accounting information. The clearinghouse could perhaps publish this information with each payment tagged with an id that is unique and only known to each user. The users can check for correct behavior as long as the clearinghouse presents the same accounting information to each user, because incorrect behavior would show up as missing payments to some artist from some user. This strategy puts some burden of verifying the transactions on the user.

The user must also verify that the songs that he listens to are labeled correctly. The clearinghouse has only the category information that is embedded in the URL. If the user wants the correct artist to be paid, he must be diligent in stopping the playing of music that is incorrectly categorized or labeled. Again, this strategy puts some burden on the user to be monitoring what the software is doing on his behalf.

Although the user is burdened with the responsibility of verifying the transactions, ultimately it belongs nowhere else. The user is spending his money, and is therefore responsible for verifying that he is getting good service for his money.

Chapter 7

Future Work and Conclusion

In a sense, a directory hierarchy is a form of distributed index. It is an index that is split up along semantic meaning, which can be arbitrarily fine grained. This pushes the splitting of the index onto people and programs, which puts larger burden on them to properly catalogue information, but it pays off in that queries can be focused on only the type of information that is of interest.

We presented directory hierarchies, a method of categorizing and retrieving data in peer-to-peer systems. We found it has many interesting properties that makes it useful for systems like Melody. It can organize content into semantic categories. It allows users to browse by category to find new content. We discussed Autocat, a necessary support program for automatically creating useful directory hierarchies. Finally, we discussed Paynet, a mechanism for contributing to artists.

Melody was the first time Chord was used in an application on a large distributed testbed such as PlanetLab. The performance differences between running on a local testbed and running on PlanetLab were very illuminating as to the difficulties in developing peer-to-peer systems on wide-area networks. PlanetLab is also a moving target for benchmarking. The composition of nodes and the latency and available capacity of the network links between these nodes is constantly changing.

7.1 Future work

There are many opportunities for improvement in Melody and the DHash layer. One outstanding issue with DHash is properly handling network partitions. A Chord ring can function even when split into two rings. If two copies of a directory block are updated independently, and then the rings rejoin, the directory blocks may both have unique data. In principle, the directory blocks can be merged into one, since none of their data is position dependent.

Other enhancements include having Melody select among replicas to choose the ones with the lowest latency or highest available bandwidth. This would improve transfer speeds by enabling Melody to store or retrieve from several replicas in parallel. Improvements to Chord RPC congestion control are also warranted.

Melody could be enhanced by the addition of per-user views of the directory structure. This would permit users to customize the hierarchy as they see fit, and share their changes with other users. Users could decide how much they trust other users before accepting their "view" of the directory structure. This would give users additional motivation to improve the categorization of the songs, and reduce the threat of malicious users adding bad data to the directory hierarchy.

7.1.1 Autocat

There are many possible improvements to Autocat, including enhancements to the filename parsing, better methods of matching similar names and using data from new sources such as directory names and prior searches. Directory paths could be examined for possible genres, since many users organize their MP3 files this way. Many songs by the same artist end up in vastly different categories due to the many different genres that people have chosen in Freedb. The categorization could be improved by glomming together songs from the same artist so that they end up in the same genre or genres. Voting among different genre sources could be used to pick the strongest candidate.

The number of forms that Autocat uses to try to interpret the MP3 filename is

fairly limited. A more general filename matching engine could extract more substrings from the filename and attempt to match each one with the names in Freedb.

In addition, name matching could be improved by using a spell-check algorithm to find partial matches in Freedb. In such a system, the dictionary would be the all the artists or song titles in Freedb, and a successful partial match would return the Freedb record for that artist or song.

Of course, both these methods depend on Freedb providing accurate and up-to-date music information. This may not be the case for newly released or unpublished songs, since users only fill Freedb with information from the CDs that they buy.

7.1.2 Paynet

Possible future work in Paynet includes implementing the SSL encryption, voluntary large-scale testing to see if users are willing to use the system and to determine if real-world song names are consistent enough to trust payments to. Actual payment could be implemented if enough people were interested.

Bibliography

[1] *Proceedings of the First International Workshop on Peer-to-Peer Systems*, Cambridge, MA, March 2002.

[2] Berkeley db. `http://www.sleepycat.com/`.

[3] Sanjit Biswas, John Bicket, Jonathan Kelner, Anjali Gupta, and Robert Morris. Starfish: Building flexible and resilient data structures on untrusted peer-to-peer systems. submitted to *Second International Workshop on Peer-to-Peer Systems.*

[4] Ian Clarke, Oscar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.

[5] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM SOSP*, pages 202–215, Banff, Canada, October 2001.

[6] FIPS 180-1. *Secure Hash Standard.* U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, April 1995.

[7] Freedb. `http://www.freedb.org/`.

[8] Gnutella. `http://gnutella.wego.com/`.

[9] Google directory. `http://directory.google.com/`.

[10] Kazaa. `http://www.kazza.com/`.

[11] Jinyang Li, Boon Thau Loo, Joseph M. Hellerstein, M. Frans Kaashoek, David Karger, and Robert Morris. On the feasibility of peer-to-peer web indexing and search. *Second International Workshop on Peer-to-Peer Systems*, February 2003.

[12] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can heterogeneity make gnutella scalable? In *Proceedings of the First International Workshop on Peer-to-Peer Systems* [1].

[13] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the First International Workshop on Peer-to-Peer Systems* [1].

[14] David Mazières. A toolkit for user-level file systems. In *Proceedings of the USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.

[15] MPEG audio layer-3. `http://www.iis.fraunhofer.de/amm/techinf/layer3/`.

[16] Napster. `http://www.napster.com/`.

[17] OpenSSL. `http://www.openssl.org/`.

[18] Overnet. `http://overnet.com/`.

[19] Paypal. `http://www.paypal.com/`.

[20] Planetlab. `http://www.planet-lab.org/`.

[21] Sean Quinlan and Sean Dorward. Venti: a new approach to archival data storage. In *FAST 2002*, January 2002.

[22] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Sun Microsystems, Inc., August 1995.

[23] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications.

In *Proceedings of ACM SIGCOMM*, pages 149–160, San Diego, California, August 2001.

[24] XMMS. `http://www.xmms.org/`.